UNIVERSITÀ DI PISA

Electronics and Communications Systems

Design of Perceptron digital circuit

Francesco Taverna

Academic Year
2023/2024

Index

Introduction

A neural network link that contains computations to track features and uses Artificial Intelligence in the input data is known as Perceptron. This neural links to the artificial neurons using simple logic gates with binary outputs. An artificial neuron invokes the mathematical function and has node, input, weights, and output equivalent to the cell nucleus, dendrites, synapse, and axon, respectively, compared to a biological neuron. Thus, Perceptron circuit provides a hands-on way to demonstrate the principles of neuron operation and also allows you to explore basic Boolean logic functions.

Whether biological, electronic, or software, the basic operation of a neuron is fairly simple at a high level. Of course, a biological neuron is quite a bit more complicated than the others. A neuron receives one or more input signals and produces an output signal.

An important part of the circuit is the activation function, which is also called a "squashing" function in neural network jargon, is usually nonlinear and also serves as a limiting operation to keep the output signal in a bounded range, in this case between 0 and 1, hence a binary output.

Algorithm description

Perceptron implements the following algorithm:

- the network has n inputs $(x_1 \dots x_n)$ represented with b_x bits
- each element x_i is multiplied by w_i, represented on $b_{w,}$ bits, that is called weight: indeed, another parameter of the circuit is the weights vector that gives a certain importance to each element of the input vector x

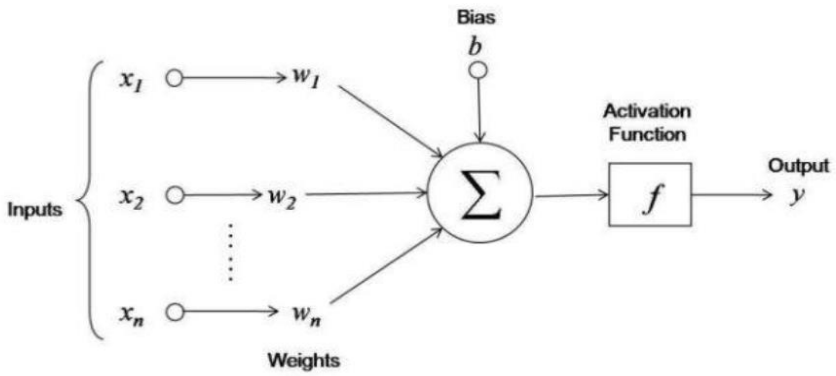

Figure 1: Perceptron scheme

- every multiplication is added with the others and a $bias\ b$ is added to the result on b_w bits
- x_i, w_i and b are considered in the range $[-1,1]$ using fixed-point arithmetic
- after adding a $bias$, the resulting value is the input for the following activation function:

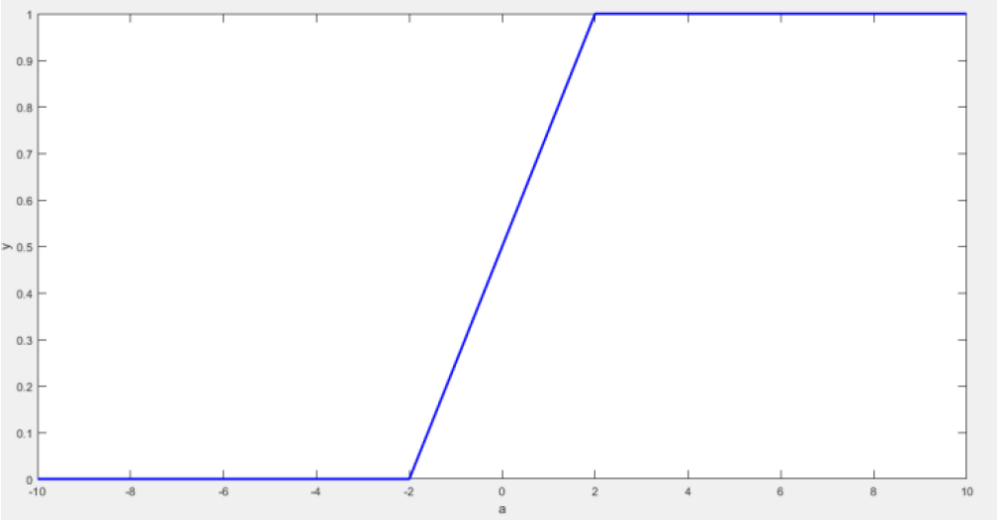

Figure 2: Activation function plot

$$y = f(\sum_{i=1}^{N_{in}}(x_i w_i) + bias)$$

$$\text{The activation function } f \text{ is defined as } f(a) = \begin{cases} 0 & if\ a < -2 \\ 1 & if\ a > 2 \\ \frac{1}{4}x + \frac{1}{2} & otherwise \end{cases}$$

- The output y of the system is represented with b_{out} bits, truncating the least significant bits.

Possible applications

The Perceptron algorithm is used a lot in Artificial Intelligence but not only. In the continuous evolving world of AI this kind of circuit can be used for implementing a binary classification, namely can be trained to distinguish two input classes, or can be trained to approximate correctly a certain dataset through its output, tuning weights vector. In particular, it belongs to the neural network context where a possible application is Multi Layer Perceptron (MLP), a complex neural network composed by multiple layers with multiple nodes that allows to carry out non linear approximations. Finally, Perceptron is used also to implement logic gates such as AND, OR, NOT.

Possible architectures

To realize an implementation of the algorithm described previously, there are 3 main components:

- A multiplier to do the products of x vector element with the corresponding of w, it's done implementing one multiplier for each multiplication that is stored in an array of products
- A chain of adder to sum the results of the previously calculated products and the $bias$. The first addition is done between the first element of product array and the $bias$ and then this sum is carried in input of another adder to sum it with the second element of product array and so on
- A circuit that implements the Activation Function of the Perceptron that is carried out through the synthesis of a specific vhdl process.

Architecture description

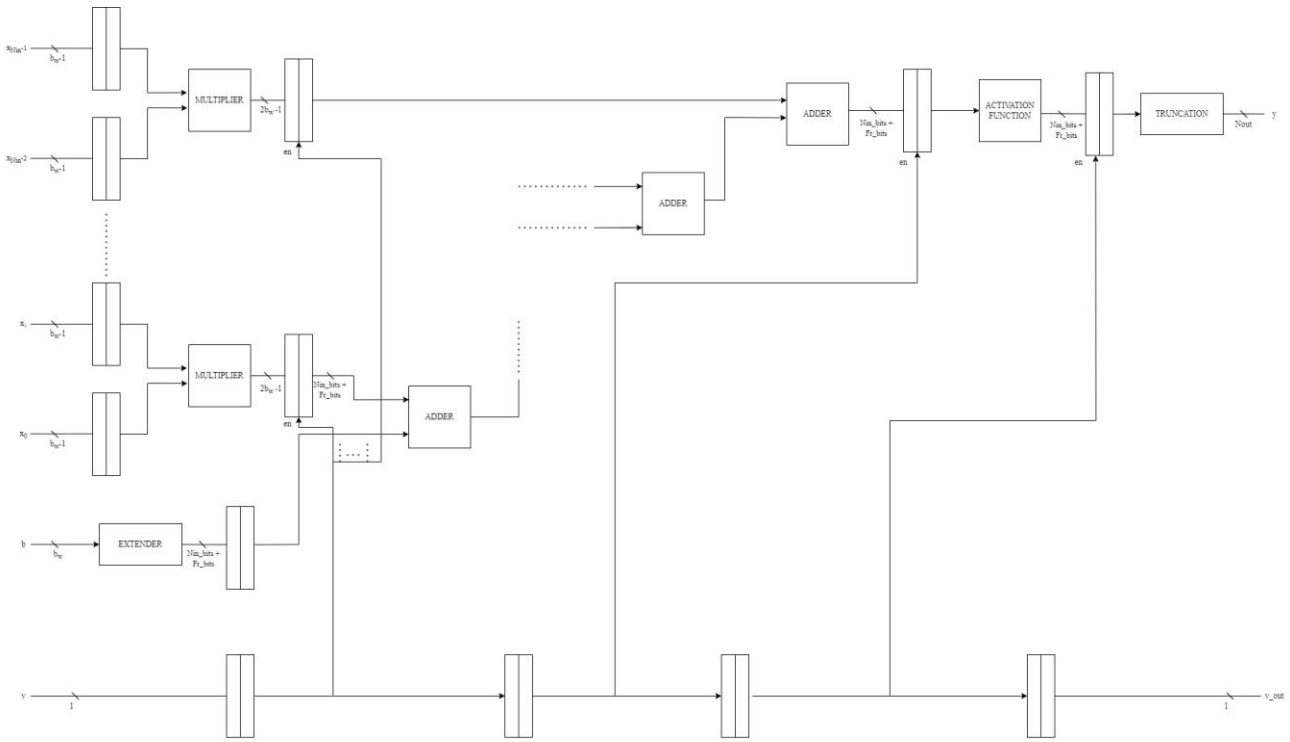

Figure 3: Perceptron logic scheme

In the previously introduced algorithm, one of the key components is the chain of adders, which computes partial sums leading to the final sum input for the activation function circuit. Preceding the addition operation there is a parallel multiplication performed by multiplier networks between elements of the input vector x and the weight vector. Another critical aspect of this architecture is the extender, utilized to increase the number of bits for the $bias$. This extension ensures that the $bias$ has the same number of bits as the other element involved in the sum operation. Although not explicitly depicted in the figure above due to space constraints, an extension operation is also implemented after the multiplication. This extension increases the output to *Nin_bits* + *Fr_bits*, aligning it with the total number of bits required to represent the final sum but the discussion about values of these constants are held later on . This is done to mitigate potential issues related to word length discrepancies.

Throughout the architecture design, a guiding principle has been adhered to, ensuring a consistent structure of paths: register-logic-register. This deliberate choice allows Vivado, during the synthesis phase, to comprehensively assess all paths and provide accurate estimations regarding timing and delays for each one.

VHDL code

In the following sections are shown the most relevant part of the vhdl code developed for describing perceptron circuit.

Note: All the cited constants are organized in *utility.vhdl* that is a package used in other modules.

Perceptron

The perceptron module contains all the components needed to implement the algorithm: it takes in input vector x of *x_arr* type, that has *N_IN* elements, in which each element has $BW - 1$ bits, vector w of *inputs* of *N_IN* elements but with *BW* bits, v that is a validity bit set to one when new inputs are assigned and a $bias$ of *BW* bits. In output this module has y on *NOUT* bits, that is the result of a truncation after the computations in the activation function and also a validity bit that shows when the result is reliable.

```
entity perceptron is

  port (
    clk : in std_logic; -- clock of the system
    reset : in std_logic; -- Asynchronous reset - active high
    x : in x_arr;
    -- each element taken from x has :
    -- integer part: 2 bits --> because range is [-1,1], so in 2's complement I need 2 bits
    -- fraction part : 6 bits --> 6 data bits (Nbit-integer part bits)
    w : in inputs;
    -- each element taken from w has :
    -- integer part: 2 bits --> because range is [-1,1], so in 2's complement I need 2 bits
    -- fraction part : 7 bits --> 7 data bits (Nbit-integer part bits)
    v : in std_logic_vector(0 downto 0); -- validity bit
    b : in std_logic_vector(BW - 1 downto 0); -- bias on BW bits
    y : out std_logic_vector(NOUT - 1 downto 0); -- output on NOUT bits
    v_out : out std_logic_vector(0 downto 0) -- validity bit in output

  );
```

Figure 4: Perceptron interface

```
type inputs is array(N_IN - 1 downto 0) of std_logic_vector(BW - 1 downto 0);
type x_arr is array(N_IN - 1 downto 0) of std_logic_vector(BW - 2 downto 0);
```

Figure 5: Important types definitions

In this specific implementation, each element of the vector x is consistently considered to have one bit less than the corresponding element of the vector w. The constant BW represents the number of bits in each w_i element. This design choice ensures a precise correlation between the lengths of x and w elements, allowing for easy adaptation by adjusting only the constant BW while maintaining the module's functionality.

It is crucial to recognize the division of bits into integer and fractional parts. Since input numbers always fall within the range of -1 to 1, only two bits are necessary for the integer part. This is because fixed-point arithmetic operates in 2's complement, ensuring that the integer bits are consistently set to two, regardless of the value of BW. On the other hand, the number of fractional bits varies based on BW. Consequently, the fractional part of the x element always consists of $BW - 3$ bits, while the fractional part of the w element always consists of $BW - 2$ bits.

In the Figure 4 are present some comments that refer to the case of BW equal to 9, i.e. the case indicated in the project specifications and these type of comments appear also in the next sections.

Parallel multiplications

In the illustrated process, the multiplication between elements of vectors x and w is executed. This operation iterates *N_IN* times, corresponding to the length of the x and w vectors. To achieve this, the conversion of *x_outreg* and *w_outreg* to signed format is performed. These variables are named as such because they represent the outputs of registers where the initial inputs are stored.

Despite the elements of *x_outreg* and *w_outreg* have different sizes, the multiplication process proceeds without complications. The result is obtained by summing the bits of the two elements.

```
--FOR GENERATE TO COMPUTE EACH PRODUCT
PRODUCTION : for i in 0 to N_IN - 1 generate
  --17 bits                                8 bits            9 bits
  prod(i) <= std_logic_vector(signed(x_outreg(i)) * signed(w_outreg(i))); --directly on 17 bits
  -- 4 + 13 = 17 bits
end generate;
```

Figure 6: For generate for product

Adder chain

Figure 7 shows the for generate loop that carries out the addition chain that is implemented differentiating two cases: the first one is the first iteration, where $bias$ is added to the first element of vector *prod_outreg*, that is storing the result of each previous multiplication after being saved in a register, and the second one that represents a generic incremental sum. It's important to note that the $bias$ addition can be done without a specific order. Each element of the sum is on the same number of bits obviously, that is *NIN_BITS* + *FR_BITS*: the extension of prod vector elements is achieved through *GenericRegisters* module that in addition to saving it, it extends it. *NIN_BITS* constant is the minimum number of bits for representing *N_IN * 1 + 1* (1 is added because there is also bias) that is the maximum positive number that this circuit can yield, the negative extreme case, i.e. *N_IN * (-1)* is covered because in 2'complement negative numbers have one more element. *FR_BITS* is minimum number of bits for representing the fractional part after multiplication process.

```
--FOR GENERATE TO COMPUTE THE SUM
SUMMATION : for i in 0 to N_IN - 1 generate

  SUM1 : if i = 0 generate
    sum(0) <= std_logic_vector(signed(prod_outreg(0)) + signed(b_extended)); --sum(0) will take the first value of the sum
    --18                                  ext 18(repeat MSB)              18
  end generate;

  SUMi : if i > 0 generate
    sum(i) <= std_logic_vector(signed(sum(i - 1)) + signed(prod_outreg(i))); --sum(Nin - 1) will take the final value of the sum
    -- 5 e 13
  end generate;

end generate;
```

Figure 7: For generate for sum

Activation function process

In this process is implemented the activation function which deals with relating the final sum of the chain to a specific output according to the linear function shown in Figure 2 that in short has three ranges: between -2 and 2 has a linear trend, before -2 the output must be 0 and after 2 must be 1. This is done putting *totalsum* signal, that stores the value of final sum, in the sensitivity list of the process and through if statements it is compared to *MINUS_TWO_IN_FPA* and *TWO_IN_FPA* that are two constants representing -2 and 2 in fixed point arithmetic. If *totalsum* is between -2 and 2 is computed the linear function and the result is set on *NIN_BITS + FR_BITS* that is the minimum number of bits in which it can be represented, 18 in the specific test case considered.

```
-- COMPUTING ACTIVATION FUNCTION
activation_function : process (totalsum)

begin

  if to_integer(signed(totalsum)) < to_integer(signed(MINUS_TWO_IN_FPA)) then
    y_toRegister <= ZERO; -- 18 bits
  elsif to_integer(signed(totalsum)) > to_integer(signed(TWO_IN_FPA)) then
    y_toRegister <= ONE; -- 18 bits
  else
    y_toRegister <= std_logic_vector(to_signed(to_integer((signed(totalsum) / 4) + signed(SFIXED_0_5)), NIN_BITS + FR_BITS));
  end if;
end process;
```

Figure 8: Activation function process

Final output assignments

Finally, are taken the *NOUT* Most Significant Bits from *y_out* signal, the output signal of the last register, and they are assigned to the real output, *y*. Last but not least is also set *v_out* in order to specify that the current output can be read.

```
--FINAL OUTPUT
y <= y_out(NIN_BITS + FR_BITS - 1 downto NIN_BITS + FR_BITS - NOUT); --17 downto 2, thus 16 bits
v_out <= v_outreg_four;
```

Figure 9: Final output assignments

Perceptron_Wrapper

Perceptron_wrapper module has been realized because the board on which the design will be put has only 100 I/O pins so it's decided to make all weights and $bias$ constant, otherwise the number of input and output bits of the network would have been 199. Thus, in utility package have been added two constants called *CONSTANT_WEIGHTS* and *CONSTANT_BIAS* set to 0.5 and they are provided as input to the perceptron module. The other reason behind this choice is that this circuit most of the times has to solve specific problems so weights are known and can be set consistently.

```
entity perceptron_wrapper is
    port (
        clk     : in std_logic ;
        reset   : in std_logic ;
        x       : in x_arr;
        -- no weights here
        -- no bias here
        v       : in std_logic_vector(0 downto 0);
        y       : out std_logic_vector(NOUT - 1 downto 0);
        v_out   : out std_logic_vector(0 downto 0)
    );
end perceptron_wrapper;
```

Figure 10: Perceptron wrapper interface

Testbenches for functional verification of the system

After description of the system is important to test if all the components and the whole circuit work correctly. The simulation is done by using two testbenches in ModelSim: one for *perceptron* and one for *perceptron_wrapper*. In both cases *BW* is equal to 9 as in project specifications and *N_IN* is equal to 10. Testbenches are designed to check if all the three parts of the activation function give correct results and also to highlight when the output y is reliable.

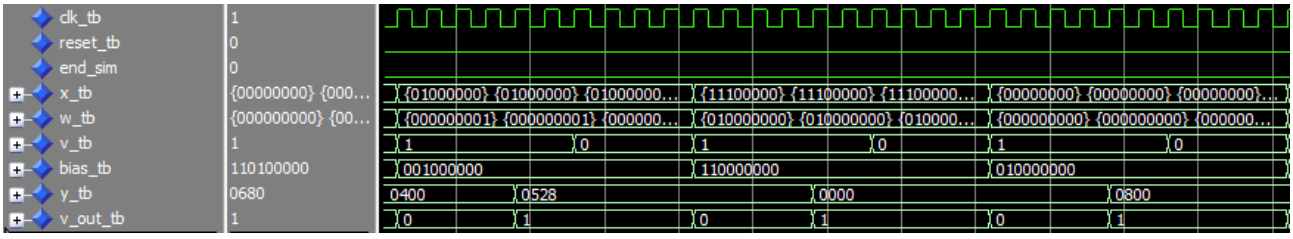

Figure 11: Testbench report of perceptron

In Figure 11 are shown three cases of the activation function:

1. All the elements of x vector are equal to 1 (01000000), all the elements of w vector are equal to 0.0078125 and b is equal to 0.5 (001000000), so the total sum, $bias$ included, is equal to 0.578125 and $f(0.578125) = 0.64453125$ that is 0528 in hexadecimal
2. All the elements of x vector are equal to -0.5 (11100000), all the elements of w vector are equal to 1 (01000000) and b is equal to -1 (110000000), so the total sum, $bias$ included, is equal to -6 and $f(-6) = 0$ that is 0000 in hexadecimal
3. The first element of vector x is equal to 1 (01000000) and all the other elements are equal to 0 (00000000), same thing for vector w and b is equal to 1 (010000000), so the total sum, $bias$ included, is equal to 2 and $f(2) = 1$ that is 0800 in hexadecimal.

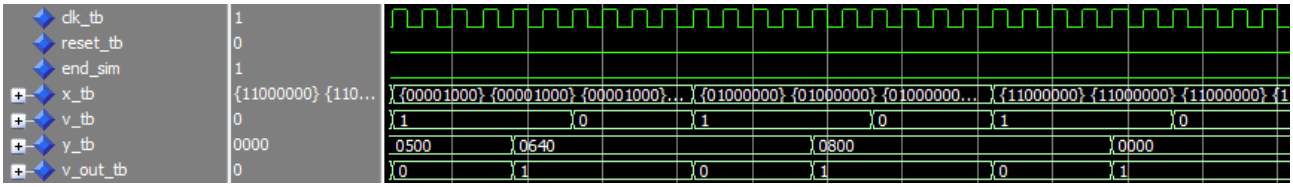

Figure 12: Testbench report of wrapper

In this other testbench, as said before, the values of w and b are fixed and are equal to 0.5 (001000000) and x is varied for testing again the three activation function ranges:

1. All the elements of x vector are equal to 0.125 (00001000), so the total sum, $bias$ included, is equal to 1.125 and $f(1.125) = 0.78125$ that is 0640 in hexadecimal

2. All the elements of x vector are equal to -0.5 (01000000), so the total sum, $bias$ included, is equal to 5.5 and $f(5.5) = 1$ that is 0800 in hexadecimal
3. All the elements of x vector are equal to 0 (11000000), so the total sum, $bias$ included, is equal to -5.5 and $f(-5.5) = 0$ that is 0000 in hexadecimal.

It's important to note that the output y becomes valid, i.e. with *v_out* equal to 1, only after four clock from the setting of inputs, this is due to the 4 registers between input and output that introduce delay. The y's value is considered valid as long as inputs are the same, when they change the validity bit is set to 0 again. This kind of solution has been chosen to maintain coherence between input and output, so output must reflect the result for that specific input.

Vivado Report

Finally, the Vivado tool was used, in order to realize and analyze the following phases:

- Elaborated design analysis
- Implementation analysis

As working device the xc7z010clg400-1 FPGA was selected.

Elaborated design

The following figure shows the Register Transfer Level description of the architecture:

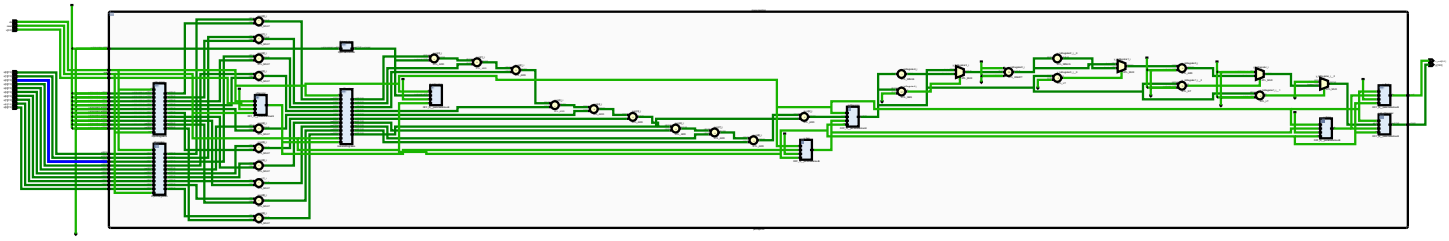

Figure 13: RTL schematic

Implementation

During synthesis and implementation phase, the modules described with VHDL are mapped to circuits that the FPGA is able to implement, exploiting the components in the Netlist. Vivado tries to do that respecting the given constraints, in this case the only one was the clock period, that, it is chosen equal to 10 ns at the beginning.

Timing Report and Critical Path

The implementation timing report is the following:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2,443 ns	Worst Hold Slack (WHS):	0,133 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	275	Total Number of Endpoints:	275	Total Number of Endpoints:	222

All user specified timing constraints are met.

Figure 14: Timing Report

With the specified constraint there is a little negative slack, so the clock period is sufficiently high in this case. Thus, the maximum clock frequency that can be obtained is around 132 MHz through the following formula:

$$f_{max} = \frac{1}{T_{clk} - WNS} \approx 132\ MHz$$

Utilization

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	Bonded IOB (100)	BUFGCTRL (32)
perceptron_wrapper	1.24%	0.63%	1.98%	1.24%	100.00%	3.13%
myperceptron (perceptron)	1.24%	0.63%	1.98%	1.24%	0.00%	0.00%
outputRegister (DFF_N__	0.00%	0.04%	0.09%	0.00%	0.00%	0.00%
outRegister (DFF_N__par	0.24%	0.03%	0.48%	0.24%	0.00%	0.00%
prodRegisters (GenericRe	0.15%	0.31%	1.02%	0.15%	0.00%	0.00%
v_four (DFF_N__paramet	0.00%	<0.01%	0.02%	0.00%	0.00%	0.00%
v_three (DFF_N__parame	0.00%	<0.01%	0.02%	0.00%	0.00%	0.00%
v_two (DFF_N__paramete	0.00%	<0.01%	0.02%	0.00%	0.00%	0.00%
vRegister (DFF_N__parar	0.00%	<0.01%	0.02%	0.00%	0.00%	0.00%
wRegisters (InputRegister	0.14%	<0.01%	0.50%	0.14%	0.00%	0.00%
xRegisters (InputxRegiste	0.40%	0.23%	1.36%	0.40%	0.00%	0.00%

Figure 15: Utilization report

Power consumption

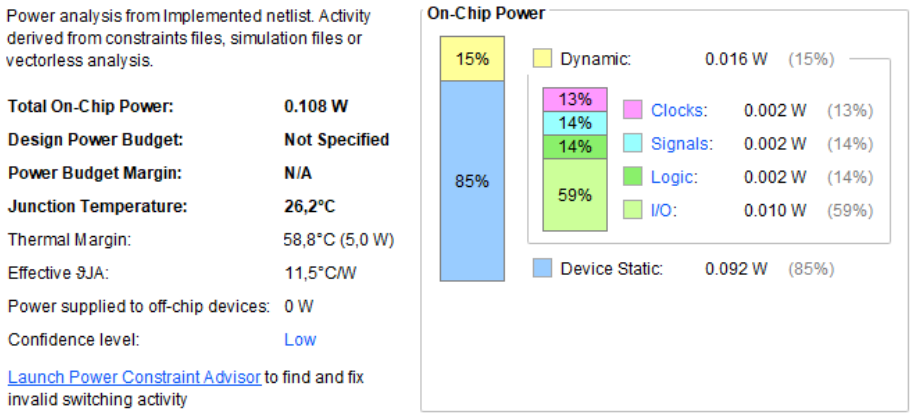

Figure 16: Power consumption report

Conclusion

In this project, it is successfully implemented and described a perceptron in VHDL, adhering to the specified parameters of $N\_IN = 10$, $BW = 9$, and $BX = 8$. One notable aspect of this design is the utilization of the generic construct to enhance flexibility, allowing for the dynamic configuration of the weight bit-width while maintaining a consistent one-bit difference between *BX* and *BW*. The choice of this approach not only fulfilled the project requirements but also introduced adaptability to accommodate various scenarios where the bit-width of weights may vary. This flexibility is a valuable feature as it allows for easy scaling and adaptation to different hardware configurations without requiring substantial modifications to the existing codebase, just need to modify *utility.vhd*. Throughout the testing phase, this implementation proved robust and reliable, demonstrating its capability to handle the specified scenario efficiently. The comprehensive testing conducted on the provided values confirmed the accuracy and functionality of the perceptron design. In conclusion, the project successfully achieved its objectives, providing a VHDL description and implementation of a perceptron and an analysis of utilization, power consumption and maximum clock frequency.